第3章 ソフトウェア開発の定石を身につけて勝ち組になろう

── 反復型プロセスとオブジェクト指向開発のエッセンス

橋本隆成

ここでは組み込みソフトウェア開発において採用が進んでいる反復型プロセスやUMLによるオブジェクト指向開発について解説する．ソフトウェアの開発プロセスは，いわば囲碁や将棋の定石集に相当する．先人の知恵を集めた開発プロセスを理解していれば，むだのない，効果的なソフトウェア開発が行える．なお，ここで紹介する考えかたは，ソフトウェア開発だけでなく，LSIの機能検証などにも適用できるだろう．（編集部）

近年，組み込みシステム開発でも「反復型プロセス」や「オブジェクト指向開発」などが積極的に導入されており，これらの開発技術や管理手法が注目を集めています．その一方で，こうした手法をじょうずに取り入れれば効果があるとわかりながらも，どのように導入すればよいのか悩まれている方も多いようです．

本稿では，組み込みシステムの開発に携わる方で，こうした手法を理解したいと考えている方を対象に，組み込みシステムに適用する際のポイントを紹介したいと思います．限られた誌面ですべてを紹介することは困難なので，ここではエレベータ・システムを例題として，反復型プロセスやオブジェクト指向開発について説明します．

1 組み込みシステムの開発手法とプロセス

現在，日本でも，組み込みシステムやリアルタイム・システムの開発にオブジェクト指向の手法が採用されるようになってきました．日本ではこれから本格的な適用が始まるところですが，海外へ目を向けると，欧米の航空宇宙産業や防衛産業では，実は1980年代から採用され，効果を上げています．日本では，つい最近まで，「オブジェクト指向開発は組み込みシステムやリアルタイム・システムに適さない」という意見が多かったため，この事実に対して意外に思われる方もいるのではないでしょうか．

● 欧米で実績のある組み込みのオブジェクト指向開発

では，日本と欧米の違いはどこから来ているのでしょう．大きな要因として，欧米では航空宇宙や軍事・防衛，原子炉制御などに代表される大がかりな制御ソフトウェアの開発プロジェクトが多数行われていることが挙げられます．すなわち，「組み込みシステム」というより，「組み込みソフトウェアを包含するリアルタイム・システム」の開発案件が多いのです．私たちがふだん開発している組み込みシステムとは少し状況が異なります．例えば，ROM/RAMなどの容量の制約は緩いと言えます．

このようなシステムは，その性質上，きわめて高い品質が要求されるのに加えて，システムが長く利用され続けます．大規模かつ高度な開発内容であるため，高品質を達成し，システムの継続的な維持・更新を可能とすることは，容易ではありません．事実，米国国防総省はソフトウェアの開発・品質維持にかかるコストの高騰が深刻な問題になると予測して，オブジェクト指向技術やAda言語の開発，CMMI[注1]などの開発と導入を推進してきました．

ソフトウェア開発を上流から効果的に進め，品質を作り込んでいくためには，さまざまなことを検討しなければなりません．そのために，オブジェクト指向技術，およびオブジェクト指向設計を効果的に実現する開発言語とプロセ

注1：CMMI（Capability Maturity Model Integration）は，米国Carnegie Mellon UniversityのSoftware Engineering Institute（SEI）が開発したプロセス能力の成熟度のモデル．SEIがCMMIを策定した当初の目的は，米国国防総省が企業にシステム開発を委託する際に，企業の開発能力を判定（監査）する手段として利用するためだった．しかしその後，多くの企業において，社内のシステムの開発プロセスを改善するための手段としてCMMIが利用されるようになり，現在に至っている．

スが必要とされています．

このように，いくつかの前提条件はあるものの，組み込みシステムやリアルタイム・システムの開発では，オブジェクト指向開発のメリットを享受できるケースが少なくありません．例えば，最近注目を集めているプロダクト・ライン・エンジニアリング注2は，組み込みソフトウェアにオブジェクト指向開発を活用して，ソフトウェア資産の再利用を促進する取り組みです．また，同じように最近注目され，実際に活用されつつあるMDA注3も，多くの事例は組み込みシステム開発やリアルタイム・システム開発であることを考えると，上述のことを納得していただけるのではないかと思います．

昨今は，日本の組み込みシステムも要求される機能が大規模になり，ソフトウェアが複雑化しています．そして，以前のようなROM/RAMなどのリソースの制約が緩やかになってきています．読者の皆さんがオブジェクト指向開発や先進的な反復型プロセスなどの開発・管理プロセスを導入するチャンスが，多くなってきたと言えるでしょう．

● 組み込みシステムのプロセスは一つではない

ひと口に「組み込みシステム」と言っても，そこには多種多様なシステムが存在します．読者の皆さんがオブジェクト指向開発やプロセスの効果を最大限に引き出したい場合，「システムの特徴に合ったオブジェクト指向開発の戦略と方法を採用するべきである」ということを筆者は強調したいのですが，実はこのことが導入の障害の一つにもなっています．というのは，組み込みシステムの仕様や開発・運用上の制約は製品ごとにさまざまで，開発手法やプロセスを画一的には扱えないからです．

その証拠に，書店などへ行っても，自分たちの開発しているシステムに適した開発手法やプロセスの情報はそう簡単には手に入りません．この傾向は，組み込みシステム開発ではとくに顕著です．忙しい時間を割いて自分たちのシステムに合わせて開発手法やプロセスをカスタマイズするのは，決して楽な作業ではありません．

開発手法やプロセスの考えかたに不慣れな方も少なくないと思います．以降で，これらを簡単に整理していきます．

● プロセスは囲碁や将棋の定石集に相当

多くのシステム開発は，エンジニアがひとりで行うほど小さな業務ではなく，普通は小規模でも数人，大規模になれば数十～数百人のエンジニアや関係者がかかわり合って作業を進めていきます．エンジニアたちは，各自がそれぞれの責任範囲の業務をほかのエンジニアとともに，時には並行して作業を進めます．

2～3人の規模であれば，きっとプロジェクトのすべての人の作業が見え，だれが何をしているのかを理解できるでしょう．しかし，メンバが10人を超えたらどうでしょう．自分がかかわる作業の範囲しか見えなくなるのが普通です．つまり，開発者の人数が多くなると，システムの開発から納入までの作業は分担して処理されるようになります．開発するシステムの一部をほかの企業に委託するとか，複数の拠点で並行開発するなど，顔を合わせて作業できないケースもあります．そうなると，各担当者が効率良く作業を進めるために，どうしても作業の手順や約束ごとが必要になります．そのために，プロセスをきちんと定義しなければならないのです(**図1**)．

プロセスには，もう一つの側面があります．それは先人の知恵を集めた定石集としての役割です．将棋や囲碁では，ルールを覚えればゲームを楽しむことができます．しかし，強くなろうと思ったら，定石集や手筋といった優れたテクニックを集めた本を読んだりすると思います．これは，今

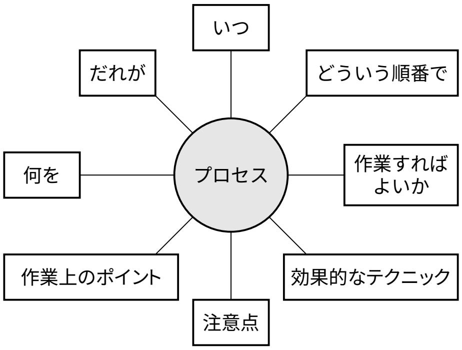


図1 プロセスとして定義されるおもな内容
各担当者が効率良く作業を進めるために，作業の手順や成果物，順守するべき規約，作業担当者，レビューの内容などが定義されなければならない．プロセスが品質と生産性を決定する．

注2：ソフトウェア集約的なシステム開発をねらい，共通で管理された一連の機能の資産を組織で共有して開発するしくみを指す．そのためには，事前に明示的かつ戦略的に，共通で核となるアーキテクチャやソフトウェア・コンポーネントなどの一連の資産(アセット)の構築と利用のインフラが重要になる．多くの場合，オブジェクト指向技術によって再利用の技術的しくみは実現される．

注3：MDA(model driven architecture)は，UMLなどによるモデルを中心として，プラットホームに依存しないPIM(platform independent model)と，依存するPSM(platform specific model)を意識しながら，ソース・コード生成までを実現する開発方法．多くの研究者やツール・ベンダが盛んにMDAのアプローチを提唱しており，近年サポートするツールも現場へ積極的に導入され始めている．

3

風に言えば，将棋や囲碁の名人たちが残した指し手や思考パターンのベスト・プラクティスです．このような定石や手筋を覚えれば，むだのない，効果的な戦いかたを身につけられるわけです．

組み込みソフトウェアのプロセスも同じです．開発作業においてどこから手をつけたらいいのか，そしてどのようなことに気をつけて作業するべきなのかなど，効果的なテクニックが紹介され，さらにコツやポイントまでまとめられているのがプロセスの価値です．具体的に言えば，要求定義の効果的な作業手順やテクニック，クラス図の描きかたや作成上のポイント，マルチタスク・システムにおけるプロセスやスレッドの分割テクニックとその指針，注意点などです．

● ウォータ・フォール型 vs. 反復型

次に，プロセスの種類について見ていきましょう．ソフトウェアは目に見えないうえに，動的なものを，ある意味強引にソース・コードという静的な記述で表現します．いきなり仕様書を見ながら，エディタに向かってプログラムをガリガリ書くことは，お勧めできません．新規開発の場合，何もないところからプログラムを作り出すまでに，大きな隔たりがあるからです．

優れた分析や設計を行うことは，実は容易ではなく，開発者の腕の見せどころとなります．そのため，実装工程前の上流の作業がきわめて重要になります．そこで要求定義や分析，設計といったように作業を段階的に進めていくことで，要求定義書とプログラムの間の隔たりを効果的に埋めていきます．

作業をどのようなフェーズに分けて進めるかについて，絶対的な基準はありませんし，開発するシステムの特徴などにも左右されます．以下では，実際の開発でよく使われている，「ウォータ・フォール型」と「反復型」という二つの開発ライフ・サイクルについて説明していきます．

1) ウォータ・フォール型プロセス

図2において，四角で表現されているのはおもな開発作業のフェーズです．多くのプロセスでは，開発作業を作業内容の特徴に応じていくつかの段階に分けて進めます．この段階を「フェーズ」と呼びます．ほとんどのプロセスは，細かな違いはあるものの「要求定義」，「分析」，「設計」，「実装」，「テスト」の各フェーズから成り立っています．

各フェーズは，作業するべき項目を明確にし，作業内容に応じて適切かつ慎重にレビューやコード・インスペクション注4などを行いながら，ソフトウェアの品質を保ちつつ作り込んでいくという意味で重要です．近年のような大規模化・複雑化しているシステム開発では，開発初期から品質を意識して開発していくことが絶対に必要です．

生産性の観点で言えば，上流工程の欠陥が下流工程で発見されて修正される場合にかかるコストや時間は，上流工程で発見・修正される場合よりも大きくなります．つまり，ソフトウェアの品質と生産性はプロセスによって決まってしまうと言っても過言ではないのです．

ウォータ・フォール型プロセスは，概念モデリング（ビジネス・モデリング）や要求定義からテストまでの作業を，明確なフェーズとマイルストーンを設けて実施するプロセスです．マイルストーンでは各フェーズの作業成果をレビューで確認します．そして，問題がなければ次のフェーズに進みます．いったん次のフェーズへ進んだら，原則的に以前のフェーズへ戻ることがないのがこのタイプのプロセスの特徴です．上流工程から下流工程へ，逆戻りすること

注4：ソース・コード中のエラーを目視などで検出する作業．いろいろな方法が提案されている．

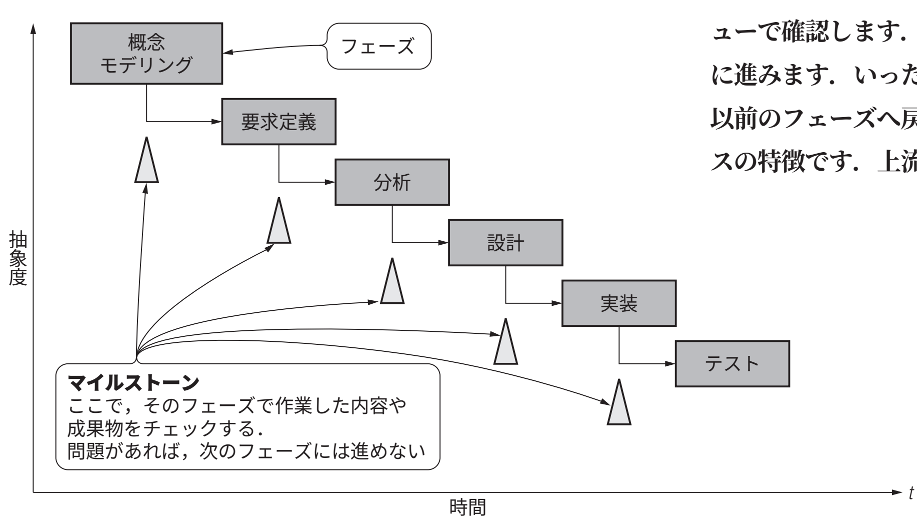


図2
ウォータ・フォール型プロセスのライフ・サイクル

明確なフェーズとマイルストーンを設けて実施するのが特徴．フェーズ間にはマイルストーンを設け，各フェーズの作業成果をレビューで確認する．いったん次のフェーズへ進んだら，原則的に以前のフェーズへ戻ることがない．上流工程から下流工程まで，滝が流れるように一方通行で作業を進めていく．

なく一方通行で流れていくというその特徴から，「ウォータ・フォール」と呼ばれています．

しかし最近では，ソフトウェアの大規模化・複雑化にともなって，新規性の高いソフトウェアの開発では，ウォータ・フォール型ではなく反復型のプロセスが主流となってきています．

2）反復型プロセス

ソフトウェアは，ハードウェアと違って直接目に見えないため，各工程間の作業成果物のレビューで，見落としやまちがいを完ぺきに発見することが困難です．また，ソフトウェアの実行速度など，性能の検証は机上のチェックやレビューでは限界があり，実際にソフトウェアを動作させてみないと確認できない検証項目も存在します．

そこで，実現する機能を少しずつ作成し，動作を確認しながら開発を進める反復型プロセスが不可欠になってきました．反復型プロセスでは，実際に機能の一部を実現したソフトウェアを開発して動作させることで，要求されている機能がユーザの要望と合致しているか，設計上の問題がないかなどを確認しながら，機能を徐々に実現していきます．

反復型プロセスは，開発のライフ・サイクルを複数回の反復（イタレーション）によって実施します（**図3**）．各反復は，原則としてウォータ・フォール型プロセスの作業をひととおり実施することで完了します．反復型プロセスは，「リスク削減型プロセス」あるいは「リスク・ドリブン型プロセス」とも呼ばれます．反復ごとにリスクを減らしていくことで，ウォータ・フォール型プロセスの問題を軽減することをねらいます（**図4**）．

反復型プロセスでは，反復ごとに，要求仕様書に記載されている機能の中から技術的リスクの高いものを優先して作業を行います．つまり，技術的課題を最初に評価することになります．反復の結果，テスト済みの実行可能プログラムが作成されるため，目標としたリスク軽減を達成できたかどうかを検証できることになります．

反復型プロセスが実際の開発に適用される際には，開発期間全体を「方向付け」，「推敲（すいこう）」，「構築」，「移行」の四つのフェーズ（ウォータ・フォール型のフェーズの定義と異なる点に注意）に分け，各フェーズの目的にあった回数，反復を繰り返すことになります（**図5**）．なお，開発期間の全体に渡って，システムを単純に機能追加していくわけではないことに注意してください．機能が反復ごとに追加されていくのは構築フェーズ以降であり，それ以前の方向付けと推敲のフェーズでは，要求作成とアーキテクチャの検討・評価に重点が置かれます．

● 構造化手法の限界とオブジェクト指向開発の台頭

次に，開発手法について説明します．開発手法はプロセスの中に取り込まれる開発上の技術です．ここでは代表的な手法を説明します．

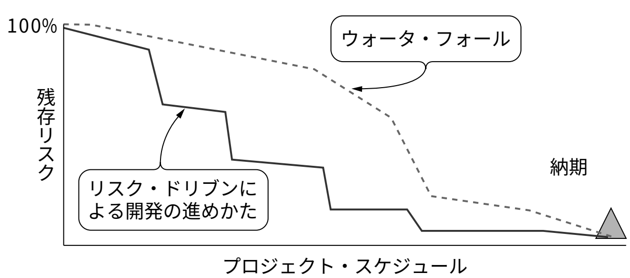


図4　反復型プロセスのリスク削減のイメージ

反復型プロセスは，反復とともに実際にアーキテクチャをテスト・評価する．そのため，技術的なリスクを反復ごとに低減していくことができる．一方，ウォータ・フォール型プロセスでは，最後のテスト・フェーズになるまで問題点が明らかにならないことがある．

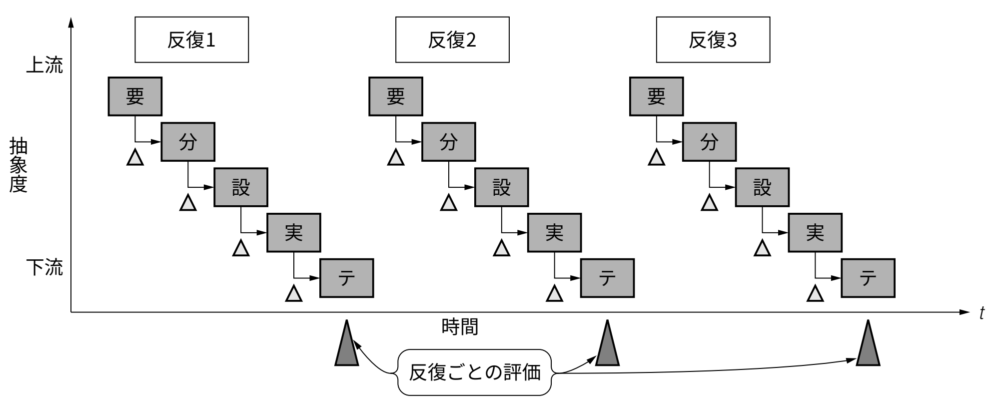
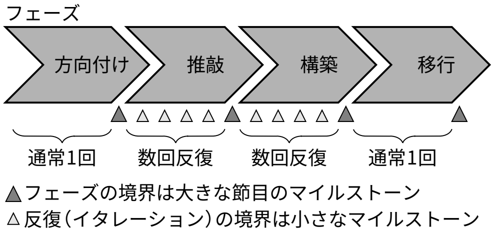


図3　反復型プロセスのライフ・サイクル

反復型プロセスでは，反復（イタレーション）を複数回繰り返すことになる．1回の反復が，いわばミニ・ウォータ・フォール型プロセスになる．

3



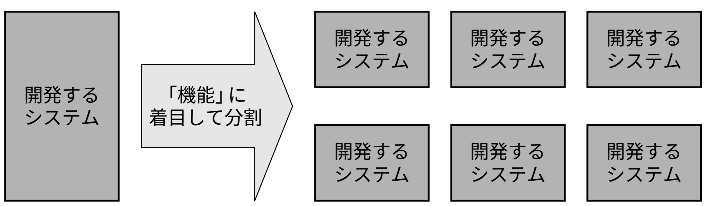
(**a**) 四つのフェーズ

フェーズ	内　容	回　数
方向付け	開発するソフトウェアの範囲や目標，開発工数の見積もり，コスト，検収条件，想定されるリスクなどを決定し，利害関係者すべてが同意する	通常1回
推　敲	大きな技術的リスクの除去がねらいである．新規性の高い開発であればあるほど，アーキテクチャの決定と評価をこの「推敲フェーズ」で行う意味が大きくなる	計画時に何回の反復かを決定する．開発期間全体の長さや，1回の反復日数を長くするか短くするかによって変わる
構　築	推敲フェーズでアーキテクチャの決定と評価が済んでいるので，反復ごとに実装し，テストを行う．反復ごとに機能が追加され，構築フェーズ終了時には機能がすべて実装される	計画時に何回の反復かを決定する．開発期間全体の長さや，1回の反復日数を長くするか短くするかによって変わる
移　行	開発したソフトウェアをリリースするに当たって，残っている作業を実施する	通常1回

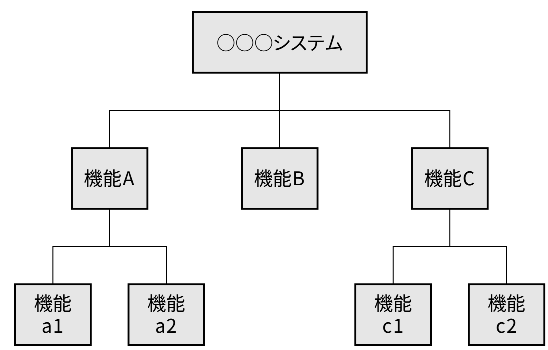
(**b**) フェーズの定義

図5　反復型プロセスのフェーズ

反復型プロセスのフェーズの定義は，ウォータ・フォール型プロセスの場合とは異なる．反復型では開発期間中に何をなすべきかを明確にし，その目的がフェーズとして定義される．通常，「方向付け」，「推敲」，「構築」，「移行」の四つのフェーズからなる．



(**a**) 機能分割



(**b**) 階層構成図

図6　機能分割による構造化手法

機能分割による構造化手法では，(a)のように，システムに要求されている機能に基づいて分割される．(b)の階層構成図は，この手法におけるソフトウェア・アーキテクチャを表現した図である．システムに要求されている機能がどのような処理に基づいて実現されるかという機能処理の点から構造化されている．

もっとも歴史があり，広く使われている手法の一つに，機能の視点に基づいて開発を行う「構造化手法(SA/SD：stractured analysis/stractured design)」があります．これは，名まえの示すとおり，「システムが行う処理」に注目して，ソフトウェアの構造を決定していく開発手法です[注5]．

20年前ほど前から少しずつソフトウェアが大規模化し，機能も複雑になってきました．それとともに，機能をベースにソフトウェアの構造が決定される機能分割の手法が登場し，普及していきました(**図6(a)**)．機能分割による構造化手法では，分析・設計作業の中でデータとその流れに着目し，DFD(data flow diagram)の作成を通じてどのような処理やデータの保存が必要かなどの検討作業を行います．そして，最終的には要求仕様書で求められている機能をどのような処理を通じて実現していくのかを決定します．つまり，ソフトウェア・アーキテクチャを処理(機能)の点から構造化し，その成果物が機能階層図として表現されることになります(**図6(b)**)．

しかし，今日では，構造化手法にはいくつかの欠点があることがわかっています．例えば，開発中に頻繁に変化する要求に対応しにくいという問題があります．また，一度開発したシステムの寿命が長く，必要に応じて機能の追加・変更作業が必要になる場合に不利になります．理由は，機能分割では機能をベースにソフトウェアの構造が決定されるので，機能変更が多くなると，ソフトウェアの構造に変更を加える必要性が出てくるためです．また，昨今は要求そのものがなかなか確定しないことが多い点を考えても，ソフトウェアの構造が機能に依存するこの手法には限界があると考えられます．そのため，現在ではオブジェクト指向開発の手法が主流となっています．

注5：開発手法の名称は，ソフトウェア・アーキテクチャである「静的な構造」の特徴に着目して付けられることが多い．例えば，「データ中心開発」，「オブジェクト指向開発」，「機能分割による構造化手法(開発)」など．それだけソフトウェアの静的な構造は重要である．

2 要求定義の作業

ここからは具体的に組み込みシステムのプロセスやオブ

ジェクト指向開発の考えかたを利用しながら解説していこうと思います．開発の進めかた，すなわち「手筋」を理解していただきます．まずは，要求定義の作業からです．

要求定義とは，開発するソフトウェアの機能要求や制約事項などを洗い出し，定義する作業を指します．要求定義の成果物は「要求仕様書」とか「要求定義書」と呼ばれています．これらは開発するソフトウェアの機能要求や制約事項などを記述したドキュメント（文書）です．

近年，ソフトウェアが複雑になるにつれて，要求定義作業が非常に困難になってきました．複雑かつ大規模化するソフトウェアの機能や制約事項を，矛盾や過不足なく正確に記述することはたいへんな作業だからです．そこで，現在ではUML（unified modeling language）規格の「ユースケース」を用いた要求定義が一般化しつつあります．ユースケースによる要求定義は，多くの優れた点を備えています．

● ユースケースでユーザ視点とシステムの要求を明確化

ユースケースは，正確には「ユースケース図」と「ユースケース記述」の二つを指しています．ユースケース図は，UMLの中で定義されているアイコンを用いて作成する，一見マンガチックな図です．一方，ユースケース記述は文章で表現される，ユースケースの詳細な機能とその使われかたの記述です．

いきなり詳細なユースケース記述を書く前に，ユースケース図を書きます．システム全体を眺めながら，どのようなアクタ（利用者，システムにアクセスする人や物）があるかを明確にし，要求されている機能を洗い出していきます．エンジニア以外の関係者も機能要求の議論に参加できるように，アイコンを利用しています．人型のアイコンをアクタ，だ円をユースケースと呼びます．

ユースケースのメリットは以下のとおりです．

- **顧客およびユーザのシステム要求の洗い出し作業において，把握する分析に漏れがないかどうかを検証するシナリオとして利用できる**
- **繰り返し型開発の単位になる**
- **システム開発上のリスク分析に利用できる**
- **繰り返し開発の計画立案に利用できる**
- **開発の単位として，エンジニアのアサイン（割り当て）に役立つ**

ユースケースを開発基本単位として進めていく方法を，「ユースケース・ドリブン開発」と呼びます．よく使うことばなので覚えておきましょう．反復型プロセスにおいてユースケース・ドリブンで開発を行う場合は，ユースケースごとに優先順位をつけ，リスクの高いものから先に作業を進めることが一般的です．ユースケースの書きかたですが，参考となる手順があるので紹介します．

【ユースケースの抽出手順】

1）アクタを探す
2）アクタがシステムとどのようなかかわり合いを持つかを考える（システムに働きかける，システムから情報を受け取るなど）
3）ユースケースがユーザの視点で見た一つの完結する処理を表すようにする
4）ユースケースが大きい場合は，複数のユースケースに分けられないかを考えてみる

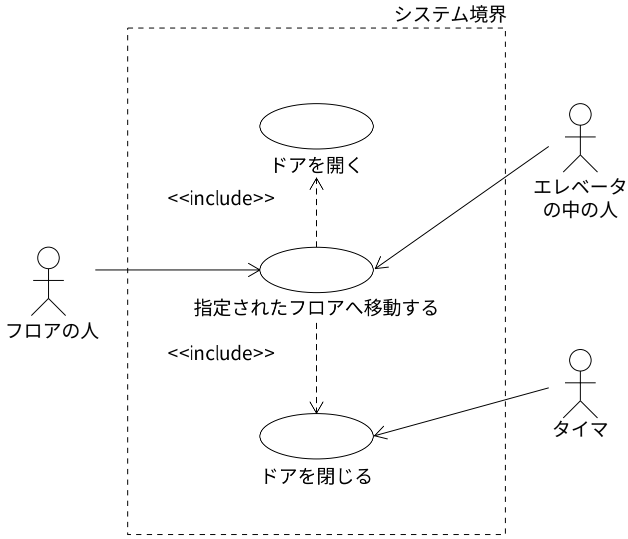
ユースケース記述では，ユースケース図によって洗い出されたユースケースを詳細に記述していきます．このとき，システムの利用をつねに意識しながら記述します．

● エレベータ・システムをユースケース図で表現

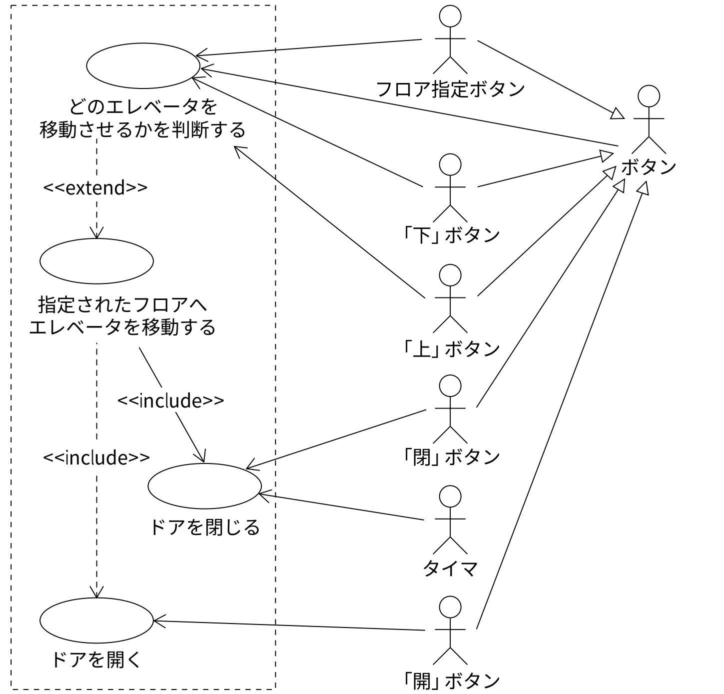
具体的な例を用いて見ていきましょう．以下に，エレベータ・システムのユースケースの例を示します．このエレベータ・システムは，「5階建てのビルで2基のエレベータがあり，各エレベータは定員オーバを判断する過重センサと，一定時間経過すれば自動的に閉じるドアがある」というオーソドックスなものです．

図7はエレベータのユースケース図です．**図7(a)**はユーザ，つまりエレベータの利用者から見たエレベータの機能です．どのような利用者（アクタ）がこのエレベータに対して刺激を与えるかということを表現しています．ここでのポイントは，抽出する機能が利用者から見た機能であるということです．決して開発者視点の機能を混在させないでください．こうすることで，ほんとうにシステムに求められているのは何なのかを検討できます．ユーザが求めていない機能を入れたり，取りこぼしたりすることを防げます．

さて，組み込みシステムの中には人が直接操作しないシステムもあります．そのような場合，ユーザ視点のユースケースをもとに，システム的にどのような機能が必要となるのかを検討し，別のユースケースを作成（詳細化）する必要があります（**図7(b)**）．つまり，ユーザの視点だけでは不十分になりやすいのです．このときのユースケースでは，アクタは人ではなく，センサやアクチュエータなどになる



(a) ユーザ視点



(b) システム・レベル

図7　エレベータのユースケース図

組み込みシステムの中には人が直接操作しないシステムもある．(a)のユーザ視点のユースケースをもとに，システム的にどのような機能が必要となるかを検討し，新たにユースケースを作成(詳細化)したのが(b)である．

ことが多いようです．このように，ユーザの視点のユースケース図およびユースケース記述と，システム分析の視点のユースケース図およびユースケース記述を用意することになります．

組み込みシステムの特徴として，正常系の処理よりも異常系の処理が多いということが挙げられます．ユースケースによる定義は一見めんどうに思えますが，正常系・異常系の処理をユースケースという単位で明確に記述するので，全体像を把握しやすくなります．各ユースケースの，この正常系・異常系の処理を「シナリオ」と呼び，システムの処理のふるまいを定義していきます．

● 非機能的な要求は開発初期に検討・定義する

ビジネス系の情報システム(非組み込みシステム)の開発手法を見ると，非機能的な要求に対する記述が少ないことに気がつきます．これは，ビジネス系ではシビアなリアルタイム性や耐故障性，ハードウェア・リソースの制約などがないか，またはほとんど開発に影響を与えないからです．そのためでしょうか，多くの開発手法では，非機能的な要求は設計作業がかなり進んでから検討するように指示されます．つまり，ビジネス系では非機能的な要求は副次的要素なのです．

しかし，組み込みシステムは違います．組み込みシステムは性能重視の世界です．開発初期にこの非機能的な要求に対する分析を確実に実施することが重要となります．組み込みシステムの場合，以下のようなさまざまな制約が課せられます．

- **プログラム言語**
- **処理応答時間などのリアルタイム性**
- **利用するリアルタイムOS**
- **ROM/RAMの制約**
- **保守性や拡張性**

この後の開発作業，とくにアーキテクチャ設計や詳細設計の工程でこの分析結果が重要な役割を果たすので，要求定義において非機能的な要求を明確にしておいてください．

非機能要求はユースケースでは書きづらい場合があります．例えば，特定のユースケースに対する制約ではなく，システム全体に対する制約があった場合です．この場合は別途，「非機能的な要求(要件)」を整理した文書を作成してください．一方，特定のユースケースに対する非機能的な制約については，そのユースケース記述に書きましょう．

3 分析の作業

次に分析作業について述べます．分析とは，開発対象について，具体的な設計で必要となる情報を抽出し，整理す

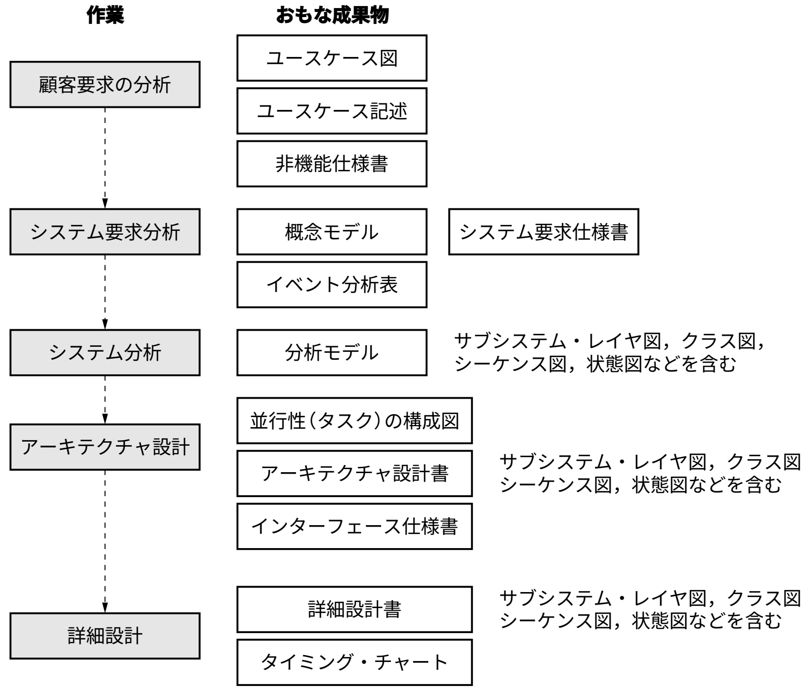


図8
作業の流れとおもな成果物の例
上から下に向かって作業を進めるのが原則で，成果物も詳細化される．作業順序関係については，並行的に実施されるのが現実である．1回の反復作業の中でも必要に応じて作業を繰り返す．

ることです．この情報の抽出と分析ですが，分析といっても実はいくつかの作業があります．具体的には，「顧客（ユーザ）要求分析」，「システム要求分析」，「アーキテクチャ分析」となります．

注意していただきたいことは，分析作業の範囲や内容の定義が，手法の提案者や企業慣習などによって少しずつ異なることです．読者の方もこのあたりを意識して分析することが必要です．

● 分析作業では開発対象の構造とふるまいを把握する

「分析」とひと口に言っていますが，実は総称的なことばとして用いられています．例えば，少し詳しく見ていくと，「要求分析」，「システム分析」，「イベント分析」などのいくつかの作業が存在します．開発手法やプロセスにもよりますが，基本的な作業の流れは**図8**のようになります．ここでは便宜上，各作業がウォータ・フォール型のように直線的に進むかのように描いていますが，実際には各作業は並行して進みますし，1回の反復の中でも必要に応じて作業を繰り返すことがあります．また，実際の成果物はテスト計画書やテスト成績書など，このほかにも多数あります．

分析作業の目的をひと言で言ってしまうと，「開発対象の構造とふるまいを把握する」ということです．開発するシステムの要求機能や制約事項を整理し，実現する対象の構造やふるまいの特徴を洗い出し，抽象化します．ソフトウェアで実現するドメイン（問題対象領域）を，論理的に「静的な構造」と「動的な構造」の両面に形式化します．

分析作業で重要なことは，開発の実行環境（OS，ハードウェア環境）や開発言語などを考慮せずに，ドメインの要求機能や制約事項を形式化することです．具体的な実行環境や開発言語に合わせた最適化は，次の設計の段階で行います．この分析作業が不十分なまま設計作業に進んでも，優れた設計は行えません．保守性や拡張性を持つシステムのアーキテクチャを設計するには，分析作業でしっかりとシステムの構造やふるまいの特徴を形式化し，把握しておく必要があります．

この分析作業は，最近，とても重要視されています．いくら要求定義が正確にできていても，大規模化・複雑化するシステムをいきなり詳細に設計することは不可能だからです．そのため，どのようにシステムを開発すれば効果的かをこの分析作業を通じて検討します．

● イベント分析で外部とシステムのかかわり合いを検討

組み込みシステムでは，外部からのイベントに対して応答し，処理を実行するために，イベント分析が重要です．イベント分析は，オブジェクト指向開発が普及する以前から組み込みシステム開発の現場で実施されていました．こ

表1
イベント分析表
イベント分析は，組み込みシステムの開発で用いられてきた．開発するシステムへの外部からの刺激をこの表で整理する．この表の情報は機能分析やタスクの並行性の設計などに利用される．

イベントID	イベント名	処理内容	発生元（アクタ）	入力・出力	同期・非同期	時間制約	備　考

こでお話しする作業内容もまったく同じです．イベント分析は，リアルタイム性が強く要求されるシステムでは，とくに重要になります．イベント分析では，組み込みソフトウェアで実現しなければならない機能のトリガとなる外部イベントやタイマ割り込みの整理を行います．これらの結果はユースケースに反映されます．

イベント分析は，システム・レベルのユースケースを作成するときに役立ちます．そのため，必要に応じて，ユースケースの定義と並行して作業を進めます．イベント分析で着目する外部イベントとしては，ハードウェアやタイマなどの割り込み，OSからのディスパッチ，ほかのソフトウェアからのイベントなどが該当し，これらがアクタとして登場することになります．

イベント分析表の例を**表1**に示します．基本的な手順は以下のようになります．

【組み込みシステムの要求分析の手順】
1）要求仕様書からユーザの要求を把握するためにユースケース（機能）を抽出する（ユーザ視点のユースケース）
2）システムに要求されているふるまいの時間的制約やタイミングを洗い出すため，イベント分析を行う．イベントの発生元がアクタに該当することが多い．

● 最初に概念モデルとシーケンス図を作成

次に，概念モデルについて説明します．**図9**に概念モデルの例を示します．

ユースケースは機能について整理・定義したものでした．一方，概念モデルは，これから開発しようとしているシステムのソフトウェア構造を分析するための最初のクラス図です．この作業は，いわゆるオブジェクト指向によるアプローチです．いきなり詳細なクラス図を作成することは困難です．そこで，開発対象（ドメイン）を理解する意味も含めて，クラスで大ざっぱにモデリングします．ドメインに登場するクラス（候補）を洗い出すことで，本格的な分析作業に必要な情報が得られます．

初めて概念モデルを作成する方の多くは，組み込みシステム特有の情報を早くから書き込みがちです．しかし，このとき，いきなり詳細なクラス図を作成してはいけません．詳細化の作業は，理にかなったシステマティックな手順を踏んで行われます．概念モデルは大ざっぱなモデルですが，軽視せずにきちんと作成してください．

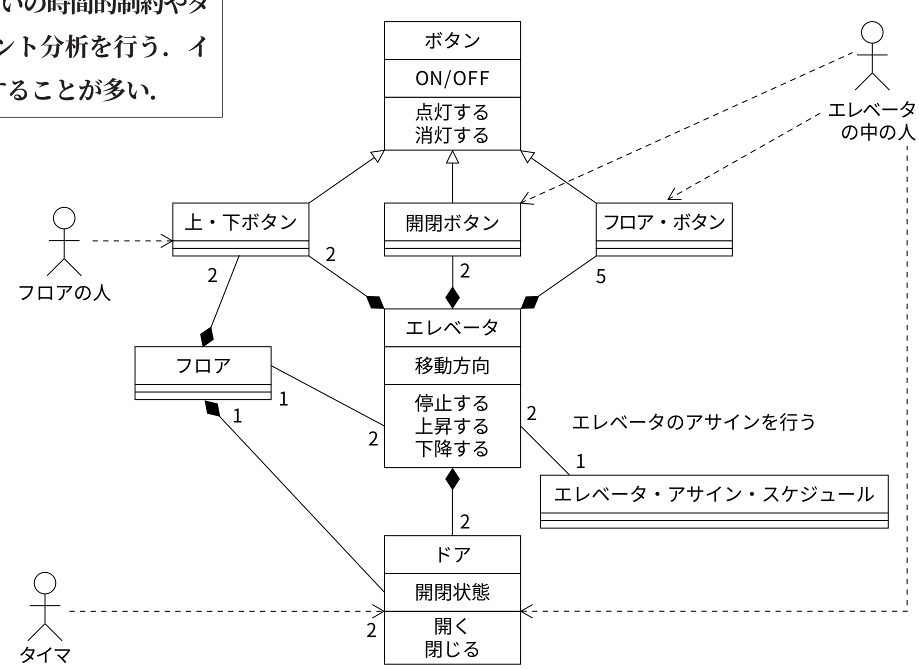


図9
概念モデル（クラス図）
開発対象（ドメイン）を理解し，ドメインに登場するクラス（候補）を洗い出すことで，本格的な分析作業に必要な情報が得られる．

さて，概念モデルができ上がったら，次に（または並行して）作成するのがシーケンス図です．**図10**にシーケンス図の例を示します．

シーケンス図を作成する目的は，ユースケースのシナリオが，概念モデルに登場したクラス（オブジェクト）と，どのように協調しながら処理を実現するのかを表現するためです．この作業の結果，機能（ユースケース）とソフトウェアの構造（クラス，オブジェクト）を対応付けることが可能となります．

基本的には，一つのユースケースの中の正常系と異常系のシナリオのすべてに対してシーケンス図を作成します．しかし，シナリオ全部となると，かなりの数にのぼります．そこで，まずはもっともオーソドックスなシナリオをいくつか選んでシーケンス図を作成します．このシーケンス図を作成することによって，クラス間の関連や各クラスがもつ属性，操作（メソッド）を明らかにしていきます．

クラス図を詳細に検討したとしても，システムに要求されている機能を確実に取りこぼしなくクラス図に記載することは不可能です．クラス図は機能を表現したものではなく，クラス間の関連をもとにソフトウェアの構造を表現したものだからです．つまり，オブジェクト（クラス図）と要求仕様書（ユースケース）は直交の関係にあると言えます．

この直交性は反復型プロセスのメリットであり，このことが非常に大きな意味を持ちます．というのは，開発するシステムの要求機能が厳密に定義されていなくても，初期のクラス図（概念モデル）を作成できるからです．そして，実際の開発では，ライフ・サイクルの初期の時点では，要求が明確に固まっていないことが普通です．

● 反復ごとにユースケースを実装，クラスを詳細化

反復型プロセスでは，1回の反復ごとにユースケースを1個～数個選び出して，実装していきます．すべてのクラスやクラスの中の属性，操作（メソッド）がいきなり全部登場することはありません．反復ごとにクラスが徐々に増えていき（減る場合もある），各クラスの情報が詳細化されます．このとき，クラスが状態を持つかどうかを検討し，状態を持つものについては状態図（ステート・チャート）を作成していきます．状態図は，組み込みシステムの開発者にとってはなじみの深いものだと思います．また，これ以外にも，詳細化されたクラス図の情報からシーケンス図を作成するなど，必要に応じてクラス図，シーケンス図，状態図に対して情報を付加しながら詳細化を進めます（**図11**）．

4 設計の作業

設計の目的は，分析作業の結果をもとに，開発の実行環境（OSやハードウェア環境）や開発言語などを考慮して，ドメインの要求機能や制約事項を最適化することです（**図12**）．この設計作業の中心は，アーキテクチャの決定と構築になります．アーキテクチャの決定では，プログラム実行時の性能やコード・サイズ，利用するミドルウェア，ライブラリなども重要な課題となります．アーキテクチャの保守性と拡張性を検討するのも設計段階の作業です．当然，OSやミドルウェアなどの専門知識を要する作業とな

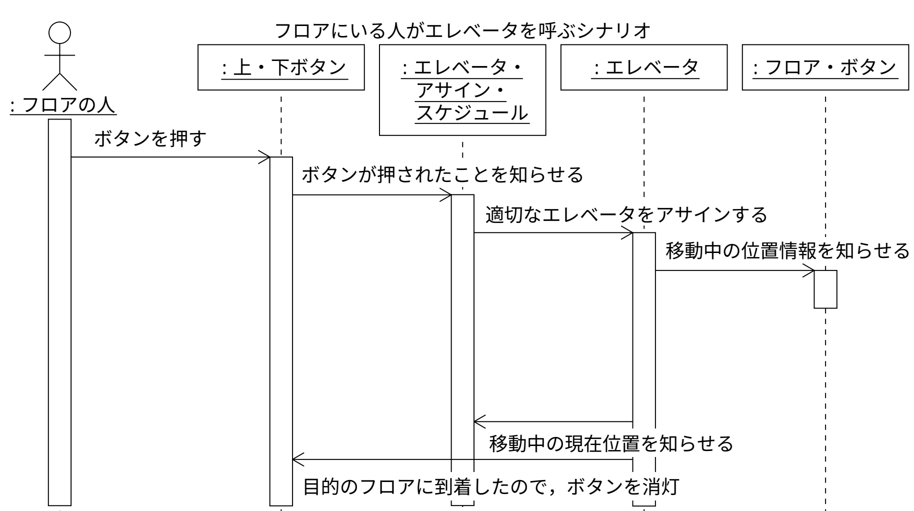


図10
シーケンス図の例
ユースケースのもっともオーソドックスなシナリオをいくつか選び，シーケンス図を作成する．この図により，クラス間の関連や各クラスがもつ属性，操作（メソッド）を明らかにしていく．

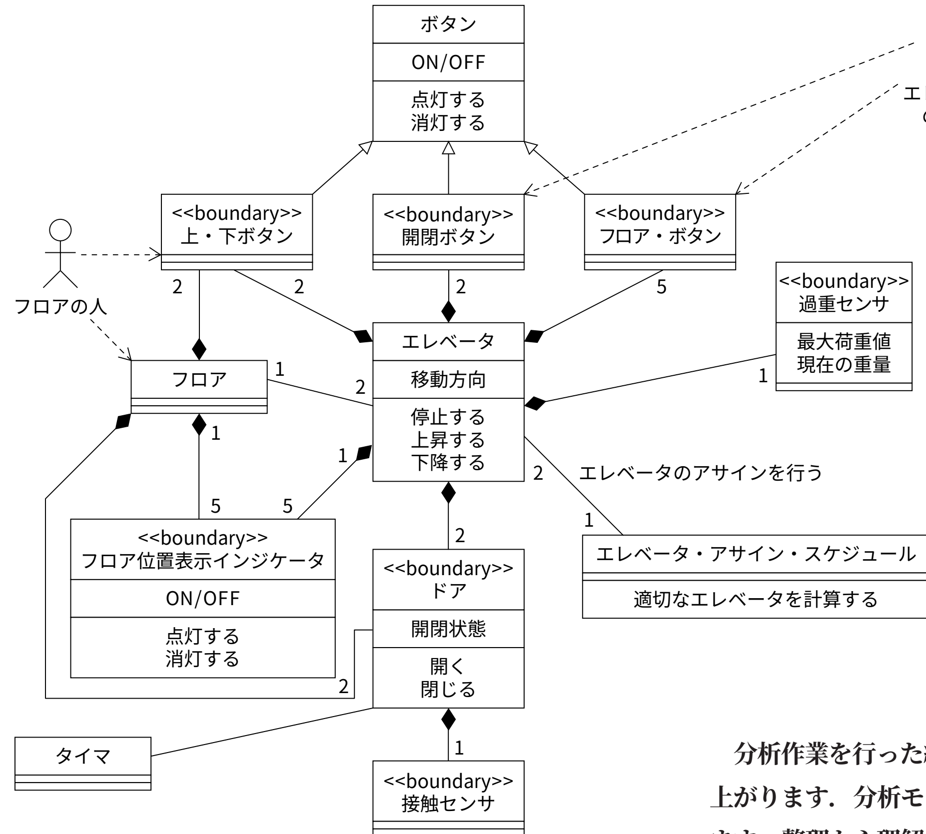


図11
情報が追加されたエレベータ・システムのクラス図

ユースケースのシナリオのシーケンス図やイベント分析の結果などを用いて，少しずつクラス図（概念モデル）を具体化していく．このとき，クラスに情報を追加したり，新しくクラスを追加したり，不要と思われるクラスを削除するなどの作業が発生する．

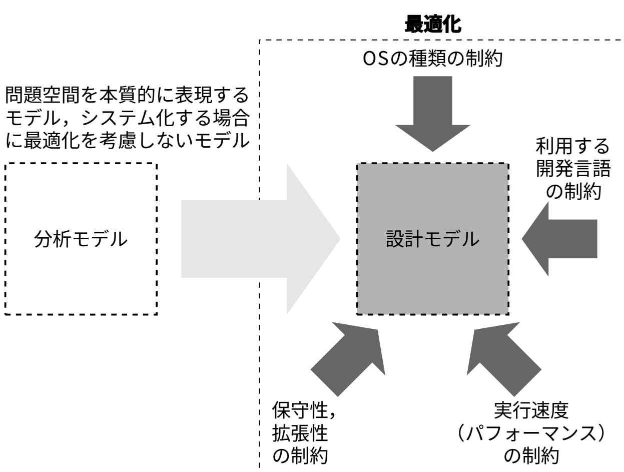


図12　設計作業と最適化

設計フェーズでは，分析フェーズの作業結果をもとに，開発の実行環境（OSやハードウェア環境）や開発言語などを考慮しながら，ドメインの要求機能や制約事項を最適化する．

り，アーキテクチャを設計するエンジニアを「アーキテクト」と呼ぶこともあります．しかし，多くの技術的課題や機能を一度に検討してアーキテクチャを構築することは困難です．そこで，反復型プロセスを用いて，実現する設計の妥当性を段階的に確認していきます．

分析作業を行った結果，成果物として分析モデルができ上がります．分析モデルの抽象度にもある程度の幅があります．整理から理解へと進んで，実装上の制約を考慮しない理想的なアーキテクチャ構成を表現することもあります．いずれの分析モデルも，あくまで分析レベルです．組み込みソフトウェアを開発する際には，さまざまな制約を考慮して，実装レベルのモデルに変換します．これを「最適化する」とも言います．

ここで言う制約とは，例えば，「開発言語では分析結果どおりに実装できない」，「実装するOS が指定されており，そのOSが提供するサービスを利用する機能について制約が生じる」，「実装環境のハード・ディスクやメモリの容量に制約がある」などがあります．加えて，前述した「非機能的な制約」なども制約になります．例えば，「可能な限り処理速度を高速にする」とか，「将来に備えて，拡張性の高いシステムにする」などは，よくある要求です．

● 静的な構造と動的な構造を決める

アーキテクチャは，「ソフトウェアの構造」を指すことばとして用いられます（学術的には厳密な定義があるが，実開発とあまり関係がないので説明を省略する）．それでは，ソフトウェアの構造とはいったい何を意味するのでしょう．

この構造には，「静的な構造」と「動的な構造」がありま す．静的な構造はたいへん重要な意味を持ち，組み込みシ

ステムを含めたすべてのソフトウェア開発で分析作業や設計作業の中心となります．

昨今のソフトウェアは，規模が大きくなったことから，アーキテクチャについて検討する際に，三つほどの粒度で議論しています．アーキテクチャの例を**図13**に示します．

いちばん大きな粒度ではシステムを「レイヤ」と呼ぶ階層に分類します．レイヤ構成については，絶対的な分けかたが決まっているわけではありません．システムの拡張性や保守性，実行時の性能などを検討し，いちばん適切なものを選びます．実際に動作させて評価しないとわからないことも多く，反復型プロセスが不可欠となってきています．

中粒度ではレイヤの中を「サブシステム（パッケージ）」という単位で構造化します．そして，いちばん小さい粒度はサブシステム（パッケージ）の中の構造化です．オブジェクト指向で開発すれば，これは一つのクラス（オブジェクト）に相当します．

アーキテクチャをこのような三つの粒度で検討するのはなぜでしょう？ 階層的な構造化には多くのメリットがあります．まず，アーキテクチャをレイヤやサブシステムで構成することで，システムの可読性が大幅に良くなります．さらに，大規模になればなるほど，複数のチームによる並行開発が不可欠になります．サブシステム間の独立性を高め，インターフェース以外の詳細な内容はカプセル化することで，サブシステム内部のクラスとクラスの間の関連などが隠ぺいされます．そして，サブシステムの利用側は，そのインターフェースのみに依存することになります．

設計者（または設計チーム）は，インターフェースの実現方法を特定する必要はありますが，外部依存性に影響を与えることなく，内部のサブシステムの設計を自由に変更できます．独立性の高いチームが集まって開発する大規模なシステムでは，これが有効に働きます．

各階層のサブシステムは，一つ下の階層のサブシステムが提供するインターフェースを利用することで，システムの修正や拡張，移植作業が必要なときに，一つのレイヤを書き直すだけで対応できます．このとき，レイヤ間のインターフェースは変更しないことが基本です．

組み込みシステムでよく見られるアーキテクチャ階層を以下に示します．

- **GUIレイヤ**：ユーザ・インターフェースなどの階層
- **ビジネス・レイヤ（ドメイン・レイヤ）**：ドメインに特有の機能を備える部分

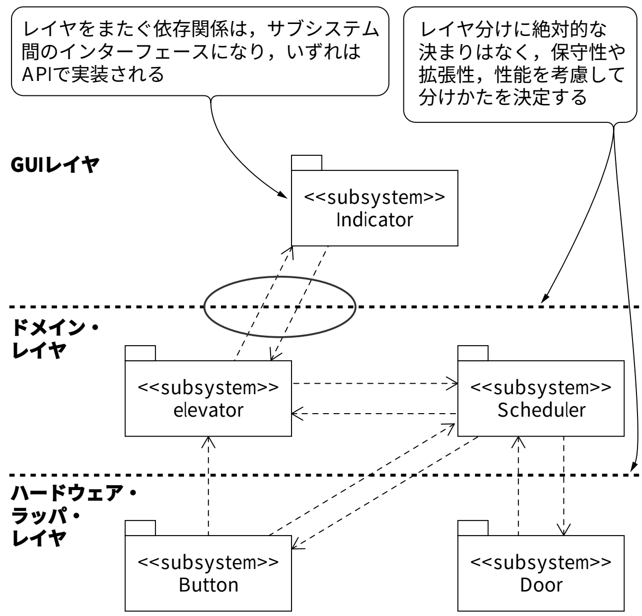


図13　エレベータ・システムのレイヤ構成の例

ここでは三つの階層に分けた．これ以外のレイヤ構成もありうる．システムに期待されている要件や制約，今後の拡張性の予定などをもとにレイヤ構成を決定する．

- **ハードウェア・ラッパ・レイヤ**：物理的なハードウェア仕様や操作手順を隠ぺいするためのレイヤ
- **ハードウェアBIOSレイヤ**：通常，アセンブリ言語やC言語で記述されるレイヤ
- **ハードウェア（物理）レイヤ**：ハードウェアそのもののレイヤ．アーキテクチャ（ソフトウェアの構造）を示すために便宜的に表現することが多い

組み込みシステムでは，さらに「動的な構造」も重要です．静的な構造と同じように，動的な構造にも粒度があります．大きな粒度の「動的な構造」としては，例えば並列・並行動作を必要とするシステムにおけるマルチプロセッサ構成やマルチタスク構成が考えられます．小さな粒度は，クラスの中の状態遷移やクラス間のメッセージのやり取りなどを決定することになります．ビジネス系のシステムの場合とは異なり，組み込みシステムは，外部からのイベントに対応した処理を行い，さらにリアルタイム性が要求されます．これを確実に実現するシステムを開発することは，技術的に高度です．

以上のことから，アーキテクチャを分析・設計することは，静的な構造と動的な構造を分析・設計することと同じと考えてよいでしょう．開発対象の特徴に合わせて，最適なアーキテクチャを採用します．アーキテクチャ設計では，

分析結果を受けて，開発システムの骨子を決定します．もちろん，新規性の高いシステム開発では，リアルタイム性などの性能やコード・サイズが制約の範囲に収まらないケースもあります．そのため，反復型プロセスに従って実際に機能の一部を作り込み，評価しながらアーキテクチャ設計と詳細設計の両方を実施します．つまり，トップダウンとボトムアップの両方から設計を確定していきます．

アーキテクチャ設計の作業内容を以下に示します．

【アーキテクチャ設計の作業内容】

1）システムの構造（静的なアーキテクチャ構造）
- システム階層（レイヤ）の設計，同一階層の分割設計
- ハードウェアなどのシステムの物理構成の決定

2）CPU仕様，CPU数，CPU間構成，バス構成
- 各物理ノード上に配置するソフトウェア・コンポーネントの決定

3）並行性/並列性の基本戦略
- 各CPUで動作させるタスクの決定
- タスク抽出の戦略決定
- イベントの送受信方法の設計
- タスク間通信の設計

4）例外処理設計

5）分散性，永続性の実現方法や製品の決定

● タスク抽出には指針がある

タスク（プロセス，スレッド）の抽出は，これまで開発者の経験的な視点をもとに実施されてきました．そのためでしょうか，実際に組み込みソフトウェア開発者に完成したソフトウェアのタスク分割の根拠を聞くと，「ベテラン・エンジニアの決定に従った」とか，「以前のバージョンのタスク分割構成に準拠した」という回答が多いようです．

タスク分割の問題は，確かに，現在のソフトウェア工学の技術では解決できていません．経験に頼るか，あるいは実際に動作させて評価しながら決定する“トライ・アンド・エラー”に依存することが多いようです．しかし，タスク分割にも参考となる基準が存在します．例えば，開発手法のDART法で有名なHassan Gomaa博士が考案した「タスク分割・統合と優先度付与」の指針が有名です．重要な点は，タスクの抽出をこの段階で行うためには，分析作業のところで「イベント分析」をしっかり実施しておかなければならないということです．イベント分析の結果をもとにした指針に従ってタスク分割とタスク間通信，およびタスクの優先度を決定します．

タスク分割の指針を以下に示します．

- **オペレータと逐次的なやり取りを行う機能/変換は一つのタスクにする**
- **ほかのタスクが利用しない資源を使う．ほかのタスクの実行結果に依存しない**
- **I/Oの処理はI/O専用のタスクにする．I/O処理以外の処理を同一タスクに含めてしまうと，タスクが長時間ブロックされる**
- **周期性がある割り込み（外部イベント），周期性がない割り込みについてタスク抽出を検討する**
- **長い時間CPUを使う処理は別タスクにする**

なお，タスクは可能なかぎり少なくしたほうが，優先度の決定などの作業が複雑にならなくて済みます．優先度が逆転するなどのやっかいな問題を避けるためにも，シンプルな構成が基本です．また，タスク数が多く，タスク間の関係が複雑だと，システムの拡張作業の際にも保守性に大きな問題を残すことになります．

● タスク分割とオブジェクトのマッピングを行う

タスク分割は，イベント分析の結果から時間制約，実行効率，タスク間の結合度などを考慮して決定されます．タスク間の通信については，原則，この段階で大まかな方針を決めます．タスク間でやり取りするデータの種類や構造も決めておくとよいでしょう．また，詳細設計の評価をフィードバックしながら，最終的なタスク間通信のメカニズム，およびやり取りするデータの内容を決定していきます．

すべてを一度に決定するのは非常に難しい作業です．そこで，抽出したタスクでオブジェクトをグループ化したり，オブジェクト間のイベントの同期/非同期を洗い出し，原則として同期イベント間のオブジェクトを同一タスクとしてグループ化します．また，グループをまたがっているオブジェクトは，複数のタスクからアクセスされるので，排他制御が必要になります．

はしもと・たかなり

<筆者プロフィール>
橋本隆成．大手国内電機メーカにて大規模リアルタイム・システムのソフトウェア開発に10年間従事．その後，大手外資系メーカ，コンサルティング・ファーム などに在籍し，ソフトウェア開発業務およびオブジェクト指向技術，開発方法論のコンサルティング業務に従事．SEI-CMMI認定コンサルタント．